Matchmaking for Covariant Hierarchies

Fabio Simeoni*
Department of Computing and Information Sciences
University of Strathclyde
Glasgow, UK
fabio.simeoni@cis.strath.ac.uk

David Lievens†
Software Structure Group
Department of Computer Science
Trinity College
Dublin 2, Ireland
david.lievens@cs.tcd.ie

ABSTRACT

We describe a model of matchmaking suitable for the implementation of services, rather than their for their discovery and composition. In the model, processing requirements are modelled by client requests and computational resources are software processors that compete for request processing as the *covariant* implementations of an open service interface. Matchmaking then relies on type analysis to rank processors against requests in support of a wide range of dispatch strategies. We relate the model to the autonomicity of service provision and briefly report on its deployment within a production-level infrastructure for scientific computing.

Categories and Subject Descriptors

D.2.11 [**SOFTWARE ENGINEERING**]: Software Architectures—*Patterns*; D.2.3 [**SOFTWARE ENGINEERING**]: Coding Tools and Techniques—*Object-oriented programming*

General Terms

Design, Algorithms

Keywords

Matchmaking, Covariance, Dynamic Deployment, Service Extensibility and Autonomicity

1. MOTIVATION

The dynamic resolution of processing requirements against pools of computational resources is a key requirement for distributed computing infrastructures. Such *matchmaking* supports the automatic composition of service resources that align with structural, semantic, and non-functional requirements [2]. Grid-based infrastructures use it to allocate clusters of storage and processors for the execution of high-performance and high-throughput applications [4]. In our experience with the design and operation of *gCube*[1], a production-level European-funded infrastructure for scientific research collaborations, matchmaking is routinely employed against software and hardware resources alike.

It is in the context of this work that we have come to consider a less conventional application of matchmaking within the local boundaries of service implementations. Here, we take processing requirements to be implicitly modelled by service requests, and interpret computational resources as software processors that implement the service interface and compete for request processing. We assume that the interface is open to covariant specialisation of input and output domains, and that the processors mirror the specialisation along one or more inheritance hierarchies. We then rely on a matchmaker to analyse the runtime types that annotate the graph structure of requests, and to infer from them a ranking of processors based on how specifically they can process the requests. Finally, we assume the application of service-specific dispatch policies over the ranking, from those that select the most specific processor to those that broadcast requests to all processors (cf. Figure 1).

We believe that the approach has non-trivial implications for the *autonomicity* of service provision. Firstly, a multiplicity of processors enables services to adapt responses to requests, but also to the observation of failures of individual processors (i.e. by dispatching to the next processor in the ranking). We thus associate local matchmaking with increased resilience and adaptability of services. Secondly, the assumption that processors may increase over time challenges the conventional expectation that service functionality remains constant after their deployment. Accordingly, we also view matchmaking as a core mechanism for the dynamic extensibility of services. In our deployment scenario, it allows us to frame third-party enhancement and specialisation of infrastructural services within a model of lightweight plugin development that promises lower costs than full-blown service development.

The rest of the paper is organised as follows. In Section 2, we illustrate the approach by example and discuss its design in more detail. We present an idealised version of the current algorithm in Section 3 and report on its deployment tests within the gCube infrastructure in Section 4. Finally, we draw conclusions and lines of further work in Section 5.

*This author is partly funded by the European Commission in the context of the D4Science project, infra-2007-1.2.2 of the FP7 IST priority.

†This author is supported by a grant from the Science Foundation Ireland.

ACP4IS'09, March 2, 2009, Charlottesville, Virginia, USA.

[1] `http://www.gcube-system.org`

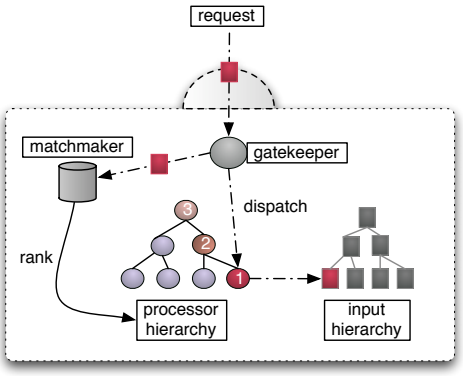

Figure 1: A 'gatekeeper' dispatches a request to the processor identified by a matchmaker as the interface implementation that is most specific to the request's input.

2. THE APPROACH

We illustrate the approach by example, choosing Java as the implementation language and leaving the technological complications of service-based implementations to Section 4. We start with abstractions for fruit and fruit processors. Fruit has a colour, processors squeeze fruit. Some processors may be specialised to squeeze only specific fruit of a specific colour. We then rely on matchmaking to identify those processors that may squeeze a given fruit.

2.1 Covariant Hierarchies

Our modelling requirements are deceptively simple. They suggest a design based on parallel hierarchies of fruit and processor classes to maximise the reuse of code that needs no specialisation in subclasses. However, they also indicate that wherever code *does* need to be specialised, the interface may need specialisation too: all processors may expose a `squeeze` method but some may refine its domain only to red pears. Essentially, the base requirement is for implementation hierarchies with covariant specialisations of input domains.

It is well known that *covariant hierarchies* are not easily reconciled with static typechecking [3]. Illustrating the problem is outside the scope of the paper though we can summarise it as follows. Covariant overriding of input domains is unsound whenever inheritance implies subtyping: a processor that overrides the *squeeze* method and specialises its domain cannot be safely used as a generic fruit processor. Covariant overloading does not compromise safety but the single-dispatch policy of mainstream languages makes it irrelevant in a polymorphic context [1]. Cast-based emulations of multiple-dispatch policies are possible, but would prove clumsy even for our simple example [5]. We prefer to de-couple inheritance and subtyping and use —what in Java is called— *generics* to base hierarchies on *parametric classes*. This inhibits unsound applications of subtyping, gives scope for generic functions, and overall makes for clearer code.

With reference to the sketch in Figure 2, the hierarchy rooted in `Fruit` is parametric in the colour type of its instances. Subclasses may well reuse code but they do not introduce new types. They introduce instead *type operators* that yield more or less specific types for the same implementation at the point of class instantiation. Subtyping is constrained to abstract over colours and is thus sound:

```
Fruit<? extends Colour> f = new William<Red>(new Red());
...f.getColour()...          //sound & allowed
f.setColour(new Yellow())    //unsound & disallowed
```

The hierarchy of fruit processors reflects a similar derivation pattern, though subclasses specialise interface and implementation explicitly, by restricting the bound over the type of fruit they handle. The internal nodes of the hierarchy are abstract and thus serve solely for code reuse; however, they remain parametric to allow further specialisation. Processors are instantiated at the leaves of the hierarchy, where classes introduce new types by instantiating the parameterisation of their parents. Again, subtyping is sound because constrained to abstract over instantiations of the type parameter.

2.2 Matchmaking

For matchmaking purposes, the static qualifications of processors must be backed up by runtime evidence. Our approach requires processors to publish a prototypical example of the fruit they can squeeze in order to provide a dynamic trace of their covariant specialisations.To this end, all the leaves of the processor hierarchy (see Figure 2) implement the `Prototyped` interface. Constructing a prototype is straightforward, though we note the conventional use of `null` to convey generality of expectations against abstract classes or interfaces (i.e. any concrete implementation will do). Later, processors may be resolved against an actual fruit and a dispatch interface, here the root of the hierarchy:

```
Fruit<?> rk = new Kaiser<Red>(new Red());
Map<FP,Float> processors = MatchMaker.match(FP.class,rk);
```

Informally, a processor matches the actual input if it implements the dispatch interface and if the dynamic type of its prototype is a *structural* supertype of the dynamic type of the input, as described in Section 3. Assuming the prototype is reliable evidence, a successful match guarantees that the actual input meets the expectations of the processor and may thus be dispatched to it. However, prototypes do not necessarily characterise processors and many may match the same input; in particular, the transitivity of subtyping implies that two processors may match the input more or less specifically. Accordingly, the `MatchMaker` quantifies matches with a *specificity score* and uses it to annotate and rank matching processors. In our example:

```
RedKaiserProcessor:   1.00
KaiserProcessor:      0.50
RedPearProcessor:     0.50
PearProcessor:():     0.16
```

`RedKaiserProcessor` and `PearProcessor` yield the strongest and weakest match, respectively. `AppleProcessor` and `WilliamProcessor` implement the dispatch interface but do not yield a match and thus do not figure in the ranking. Unsurprisingly, match correlation may remain ambiguous against the subtype lattice: the score-based ranking may hide a partial order in terms of specificity. The `MatchMaker` has no further evidence to prioritise fruit colour over type, and thus ranks equally `RedPearProcessor` and `KaiserProcessor` as second best. As shown in Section 4, disambiguation may come from the client, if and when necessary.

```
abstract class Colour {...}
class Yellow extends Colour{...}
class Red extends Colour{...}

abstract class Fruit<C extends Colour>{C c; Fruit(C c){...} C getColour(){...} void setColour(C c){...}...};
class Pear<C extends Colour> extends Fruit<C>{...};
class Apple<C extends Colour> extends Fruit<C>{...}
class Kaiser<C extends Colour> extends Pear<C>{...};
class William<C extends Colour> extends Pear<C>{...};
abstract class FP<F extends Fruit<?>> {void squeeze(F f) {...}} //abstract processors for incremental code reuse
abstract class AP<A extends Apple<?>> extends FP<A> ...}
abstract class PP<P extends Pear<?>> extends FP<P> {...}
class FruitProcessor extends FP<Fruit<?>> implements Prototyped<Fruit<?>> {   //concrete processors for matchmaking
...public Fruit<?> getPrototype() {return null;}...}
class AppleProcessor extends AP<Apple<?>> implements Prototyped<Apple<?>> {
   ....public Apple<?> getPrototype() {return new Apple(null);}...}
class PearProcessor extends PP<Pear<?>> implements Prototyped<Pear<?>> {
   ...public Pear<?> getPrototype() {return new Pear(null);}...}
class KaiserProcessor extends PP<KaiserPear<?>> implements Prototyped<KaiserPear<?>> {
   ...public KaiserPear<?> getPrototype() {return new KaiserPear(null);}...}
class WilliamProcessor extends PP<WilliamPear<?>> implements Prototyped<WilliamPear<?>> {
   ...public WilliamPear<?> getPrototype() {return new WilliamPear(null);}...}
class RedPearProcessor extends PP<Pear<Red>> implements Prototyped<Pear<Red>> {
   ...public Pear<Red> getPrototype() {return new Pear<Red>(new Red());}...}
class RedKaiserProcessor extends PP<KaiserPear<Red>> implements Prototyped<KaiserPear<Red>> {
   ...public KaiserPear<Red> getPrototype() {return new KaiserPear<Red>(new Red());}...}

interface Prototyped<T> {T getPrototype()}
```

Figure 2: Covariant hierarchies.

The input can then be dispatched to matching processors according to some policy at the client's discretion. The `MatchMaker` ignores how processors should be applied to fruit: the existence of a `squeeze` method and the details of its signature rest entirely with its clients. One possibility among many is to dispatch from the most to the least specific processor until one completes successfully:

```
for (FP processor : processors.keySet()) {
  try {processor.squeeze(rk);}
  catch(Exception tolerated){}
}
```

The invocations of `squeeze` show clearly the interaction between static and dynamic typechecking: their soundness is an implication of matchmaking at runtime and could not be guaranteed by the typechecker. In particular, we use raw types to bypass the static typechecking regime under the guard of the matchmaker.

3. THE ALGORITHM

At the heart of our matchmaker is the recursive algorithm shown in Figure 3. The algorithm is repeatedly fed with the actual input and the prototypes of processors that match the dispatch interface, as shown in Section 2. For each prototype, its task is twofold: (a) to traverse the graph structures of input and prototype and verify that the dynamic types of corresponding nodes are in the subtyping relation; and (b) to quantify with a global score in the range $(0, 1]$ all the type specialisations that may be observed at pairs of matching nodes. Prototypes that pass the dynamic typecheck identify processors that can safely consume the input, in spite of their deep covariant specialisations of the dispatch interface; the scores allow to rank processors according to how specifically they may consume the input. Prototypes that do not pass the typecheck induce type errors and their processors are excluded from the ranking returned to clients.

We capture the essence the algorithm in an ad-hoc notation that minimises the verbosity of reflection algebras, type systems, and control flow. At each call, a preliminary check enforces subtyping of corresponding nodes in the graph structures of input and prototype (`CHK`). Object graphs may well be cyclic and the algorithm detects them by keeping track of the node pairs it visits (`HYP`): optimistic assumptions precede the comparison, propagate recursively along with it, and prove right if the same node pairs are re-visited without evidence of errors.

The analysis then distributes along the fields of the prototype node with the goal to compute per-field scores and recompose them later according to some *scoring strategy*. There are cases to distinguish at each field, based on the value or its dynamic type. The first cases cover the possibility of `null` values in prototypes and inputs (`NULL` cases). [2] These carry no type information at runtime (`dtypes`) and force the algorithm to resort to static knowledge (`stypes`). [3] In all cases, structural recursion is inhibited; the algorithm then relies on the auxiliary function `distance` to measure specificity in terms of the distance between types in the subtyping lattice. The remaining cases are type-based and mir-

[2] Remember from Section 2 that `null` is a convention on the construction of prototypes that marks independence from concrete implementation of an abstract type.

[3] Incidentally, this explains why case analysis is performed on the fields of matching nodes rather than on the nodes themselves. The matchmaker adds an artificial root to the prototype and object graphs to 'push' all cases within fields.

```
match(prototype AS p,input AS i,hypotheses AS hyp)

  IF i HAS NOT p.dtype THEN ERROR                          /*CHK*/
  IF hyp.contains(p,i) THEN RETURN 1 ELSE hyp.add(p,i)     /*HYP*/

  strategy is getScoringStrategy()

  FOREACH field IN p
     fieldScore IS
       CASE i.field IS NULL AND p.field IS NULL => 1/distance(p.field.dtype,i.field.stype)      /*NULL1*/
       CASE i.field IS NULL AND p.field IS NOT NULL => 1/distance(p.field.dtype,i.field.stype)  /*NULL2*/
       CASE p.field IS NULL AND i.field IS NOT NULL => 1/distance(p.field.stype,i.field.dtype)  /*NULL3*/
       CASE (p.field.dtype IS ATOMIC) => 1                                                      /*ATOM*/
       CASE (p.field.dtype IS ARRAY) =>                                                        /*ARRAY*/
            IF p[0] IS null THEN 1 ELSE SUM_el(match(p[0],i[el],hyp)/i.length
     DEFAULT CASE => match(p.field,i.field,hyp)                                              /*BASE*/
     scoringStrategy.addFieldScore(fieldScore);

  depthScore is 1/distance(p.field.dtype,i.field.dtype)                                      /*DEPTH*/
  scoringStrategy.addDepthScore(depthScore)

  RETURN scoringStrategy.computeScore
```

Figure 3: The Matchmaking Algorithm.

ror the modelling primitives of the language. Specificity is measured recursively against object structures (`BASE`) while it is not at stake against atomic types (`ATOM`). For arrays, the algorithm exploits a second convention on the construction of prototypes: one element suffices to convey type expectations. It then averages the scores obtained by comparing such prototypical element with all the elements of the input array. This is not possible if the prototypical element is `null` as erasure leaves no type against which loss of specificity may be measured.

Field scores measure specificity along the structure of the prototype but do not reflect all the structure of the input, which may spread across additional fields. Nor do they cater for purely behavioural specialisations (additional and overridden methods). For this, the algorithm accommodates feeds a dedicated *depth score* into the scoring strategy. Again, the depth score is based on the distance between the dynamic types of the current node pair (`DEPTH`). Once all the score contributions of subcomponents are available to the strategy, the algorithm can finally compute the composite score and return it.

We factor the scoring strategy outside the algorithm to acknowledge that there are different ways to assimilate multiple scores into a single one. Different strategies capture different facets of the intuition and may be preferred in different contexts (cf. Figure 4).The `AvgStrategy1` used in Section 2 takes a full averaging approach to score composition, essentially treating the depth score on par with field scores. Averaging has two related effects: (a) scores that indicate the specificity of the prototype boost the composed score in a way that accumulates along the call chain; (b) scores that indicate generality of the prototype can balance themselves out in the composed score. Both effects may be deemed undesirable: (a) introduces a strong bias to depth and can reduce the intelligibility of the global score; similarly, (b) may hide specialisations, particularly those associated with behaviour alone. `AvgStrategy2` compensates for (a) by excluding the exact matches (i.e. those with a score of 1) from the composition. The `MixedStrategy` eliminates also some of the negative implications of (b) by averaging only on field scores and using the depth score as a penalisation factor (cf. Section 2). Like `AvgStrategy1`, the `GeometricStrategy` makes no distinction between field and depth scores. However, it also renounces averaging altogether. As all scores are combined geometrically, the implications are in fact opposite: the focus is entirely on penalising the generality of prototypes rather than rewarding their specificity. Accordingly, scores drop very quickly.

4. A CASE STUDY

We have experimented with matchmaking in the implementation of one service of the gCube infrastructure. The *DIR Master* is used to optimise the evaluation of content-based queries across a number of distributed target collections, and is just one component of a broader framework for service-based Distributed Information Retrieval (DIR) [6]. The service identifies the collections that appear to be the most promising candidates for the evaluation of a given query (*collection selection*). It also integrates the partial results obtained by evaluating the queries against individual collections (*result merging*). Both functionalities admit a range of implementations that diverge algorithmically, often in reflection of different assumptions on the structure and semantics of inputs and outputs (different queries, different results, different selection and merging criteria). For an infrastructural service, it is undesirable to bind functionality to a fixed set of implementations and to distribute it across service interfaces that diverge in the shape of inputs and outputs. It is equally unsustainable to base its adaptability to community requirements on full-blown service development. We have thus resorted to matchmaking to manage multiple implementations under a single interface that is *dynamically* open to arbitrary specialisations. We have then experimented with the dynamic deployment of additional implementations as a cost-effective development model of service extensibility.

```
abstract class ScoringStrategy {
  List<Float> scores = new ArrayList<Float>();
  void addFieldScore(float score) {scores.add(score);}
  void addDepthScore(float score) {addFieldScore(score);}
  abstract float getScore();}
class AvgStrategy1 extends ScoringStrategy {
   public float getScore() {
    float score=0;for (Float f:scores) score=score+f;return (score==0)?1:score/scores.size();}}
class AvgStrategy2 extends AvgStrategy1 {
   public void addFieldScore(float score) {if (score<1) super.addFieldScore(score);}}
class MixedStrategy extends AvgStrategy2 {
  Float depthScore;
  public void addFieldScore(float score) {if (score<1) super.addFieldScore(score);}
  public void addDepthScore(float score) {this.depthScore=score;}
  public float getScore() {return super.getScore()*depthScore;}}
class GeometricStrategy extends ScoringStrategy {
   public float getScore() {float score = 1;for (Float f:scores) score=score*f;return score;}}
```

Figure 4: Scoring Strategies.

Specialisation of service interfaces is easily accommodated with standard Web Service technologies. We have routed client requests to a single gateway and used the matchmaker to dispatch them from the gateway to suitable implementations of `Ranker` or `Merger` interfaces. The dispatch policy and the hierarchical arrangement of ranking processors, merging processors, their inputs and their outputs follow the patterns shown Section 2. As also suggested in Section 2, we have extended the `Prototyped` interface to name processors, and added functionality for name-based lookup. Processor names are published as part of the service description and clients may include them in requests for a tighter functional coupling with the service. Pre-defined processors register with the matchmaker at service startup. Additional processors may become available at runtime as the payload of *service plugins*, i.e. lightweight extensions of back-end service functionality.

Plugins are distributed in standard JARs and may be deployed within the *Master*'s implementation through dedicated operations of the interfaceBy a convention on the JAR manifests, distributions name the implementation of a `Plugin` interface which exports the functional extensions contained therein. These include the additional processors and the implementations of inputs and outputs that may be associated with their specialisations of the service interface. In particular, processors are published as new capabilities of the service and registered with the matchmaker to process future requests that carry compatible inputs. Finally, the plugin distribution is persisted and its functional extensions locally redeployed at subsequent service startups.

5. CONCLUSIONS AND FURTHER WORK

Our deployment tests indicate that type-based matchmaking may be conveniently embedded in a number of gCube services. At the time of writing, the shift from a proof of concept to a systematic adoption within the infrastructure is officially under planning. Partly, this requires augmenting development tools with abstractions that simplify and standardise the adoption of matchmaking within service implementations.Most importantly, it calls for a refinement of infrastructural mechanisms for service publication, discovery, notification, persistence, and deployment to the case of service plugins. Dependency management is particularly challenging in this scenario and may require integration with dynamic module systems such as OSGi[4]. The overall vision is of an infrastructure that can autonomically extend its capabilities by monitoring the publication of plugins and by overseeing their deployment within running services. We conclude with a speculation on the broader role of a local matchmaker within software design, particularly its potential as a general-purpose design patternThe observation here is that matchmaking combines under a dynamic framework the multiplicity of implementations associated with the factory pattern with the management of implementation dependencies associated with factory, locator, and dependency injection patterns. Even though such combination seems generally desirable, further investigation is required to assess its potential in application domains other than autonomic service development.

6. REFERENCES

[1] C. Clifton et al. Multijava: modular open classes and symmetric multiple dispatch for java. In *OOPSLA*, pages 130–145, 2000.

[2] I. Constantinescu and B. Faltings. Efficient matchmaking and directory services. In *WI '03: Proceedings of the 2003 IEEE/WIC International Conference on Web Intelligence*, page 75, Washington, DC, USA, 2003. IEEE Computer Society.

[3] R. Ducournau. "real world" as an argument for covariant specialization in programming and modeling. In *OOIS Workshops*, pages 3–12, 2002.

[4] I. Foster and C. Kesselman. *The Grid 2: Blueprint for a new computing infrastructure*. Morgan Kaufmann Publishers, pub-MORGAN-KAUFMANN:adr, second edition, 2004.

[5] P. P. O. P. and L. B. S. Raccoon. Multiple downcasting techniques. *SIGSOFT Softw. Eng. Notes*, 24(3):69–75, 1999.

[6] F. Simeoni et al. A grid-based infrastructure for distributed retrieval. In *ECDL*, pages 161–173, 2007.

[4] `http://www.osgi.org`